Simple Intuitive Models of Programming

R. C. Tausworthe
DSN Data Systems Development Section

This article hypothesizes that mathematical models of the programming process can be formulated to gauge the sensitivities of that process to various given parameters, and that such models can be calibrated on an empirical basis and used as guides toward maximizing productivity, documentation quality, and programming reliability. The article then presents three oversimplified models as illustrations.

I. Introduction

The computer is a medium of artistic expression in the hands of a creator; it permits its user to fulfill, to the maximum extent of his human capability to communicate, almost any computational desire which can be codified into a programming language. And so, programming is truly an art form for many who are afforded the luxury of such a willing servant, but with little obligation to produce something industrially useful. To those of us, however, who view the computer as a tool, an integral part of our daily lives for the accomplishment of organizational goals and for the solution of problems we were hired to solve, the "art" of programming needs to be a science, or at least an engineering discipline.

There are many of us who classify ourselves as "nonprofessional programmers," but who have used programming as a research tool (in the author's case, to probe communications-theoretic problems, such as the threshold behavior of phase-locked receivers, the design of planetary ranging systems, the performance characteristics of telemetry systems, and so forth). The power of computers to describe, model, simulate, and solve very complicated mathematical problems has, in effect, catapulted the frontiers of science, so that we can now not only predict systems performance with amazing accuracy, but also optimize systems parameters to enhance that performance.

To develop a mathematical model which portrays a system and predicts, with some degree of fidelity, its per-

formance figure of merit, is a matter of professional training and the application of that training in a disciplined approach to problem solving. Standard mathematical techniques aided by computers normally yield system parameters which produce optimum or enhanced performance, if any such parameters exist on a practical basis.

Even when systems have a stochastic element in them (such as those found in space communications), their performance can be quantified: one learns to cope with randomness on an everyday, friendly basis, because even randomness exhibits certain reliable macroscopic regularities, in spite of microscopic unpredictability. One learns what things are mathematically ascertainable about stochastic processes, and what things are not. One learns to compensate, at the design level, for the adverse effects of noise and randomness in communications channels, through characterization of the environment in which the communication must take place. Space communications is thus no longer an art, but a bona fide engineering discipline.

Computer programming itself is a stochastic process, just as much as deep-space communications is, and perhaps more. However, to my knowledge, it has not been studied as such, nor characterized as fully as it needs to be to lift it from the "art" category to the "discipline" category. Whereas computer programs have been written to describe, model, and/or simulate very complicated mathematical problems and almost every other conceivable kind of system and/or process, there does not seem to be much effort in developing a mathematical theory or computer program which explains what is knowable about the process of programming. There are, however, some models of program reliability [1,2].

Many have perhaps scoffed at the prospects for formalizing the analysis of programming, both as a mathematical possibility, as well as an enterprise from which any useful information could be gained. Nevertheless, *it is the hypothesis of this paper that mathematical models of the programming process can be formulated to gauge the sensitivities of that process to various given parameters. Such models can be calibrated on an empirical basis and used as guides toward maximizing productivity, documentation quality, and programming reliability.* .

The first steps in demonstrating this hypothesis are to characterize some of the measurable aspects of software development and to postulate how various of the parameters correlate with one another and with reality.

II. Mathematical Modeling

Models of complex systems are seldom exact, especially when there are unknown factors. In such cases, the answers obtained must be weighed against intuition and observable data. Interpretations of results obtained from a model are often used to build levels of intuition; however, one must trust intuition only insofar as it reinforces physical evidence or aids in the creation of a more adequate mathematical model, or explains results coming from that model.

At times, it will be necessary to make outlandish simplifying assumptions just to arrive at an approximate first result. Such simplicity only tends to temper how much one can believe about that which the model tells as a general truth. If the response is favorable, then more detailed theories can be sought, more complicated models developed, until answers are known in believable precision.

This article therefore develops three outlandish, simple "beginning" models for certain aspects of the programming process, based on intuitive hypotheses and intuitive "proofs;" conclusions reached are therefore only approximate truths—but truths that are refinable by extended modeling and measurement.

When more precise models of programming someday come into existence, the process of programming can be optimized more scientifically. Until then, we are stuck with using more subjective means, opinion, and "gut-feeling" intuition to better the process.

III. Productivity Model

The first model is one of programming team efficiency. Let us suppose that there are W workers in a team who have just completed and delivered a documented, bug-free program. Each worker has spent time T_i, $i = 1, \cdots, W$ in the project, and each has an individual productivity p_i, $i = 1, \cdots, W$, expressed in some arbitrary common team production units. The average time T spent by each worker in the project is

$$T = (T_1 + \cdots + T_W)/W$$

and the average individual productivity P_I is

$$P_I = (p_1T_1 + \cdots + P_WT_W)/TW$$

Individual productivities p_i are measures of how much each worker has contributed to the deliverable final product per unit of time when unencumbered by the team structure. That is, when working alone, fully informed, each worker is capable of turning out p_i units per day.

However, for a team to function, there must (of necessity) be some time spent in interfacing and coordinating activities between workers, during which, nothing deliverable is produced. This fraction of time thus constitutes a loss factor insofar as productivity goes.

Let us suppose that each worker has spent only the *necessary* fraction t_i, $i = 1, \cdots, W$, of his time in such activities, and that the remainder of his time was spent producing at his normal, unencumbered rate. The average fractional time $t(W)$ spent in non-production is then defined as the *team-interaction factor*,

$$t(W) = (p_1T_1t_1 + \cdots + p_WT_Wt_W)/P_lWT$$

The value of $t(W)$ is almost certainly an increasing function of W, since (intuitively) the more workers there are, the more likely that interfaces between individuals will exist.

The overall team production rate P_T is now given by the simple formula

$$P_T = P_lW[1 - t(W)]$$

This is a universal formula for team productivity, not merely one applicable to programming efforts. The implications one may gain from it are thus widely applicable to a multitude of management decisions concerning project organization.

A. What Lessons Can Be Learned From This Model?

It is obvious from the P_T equation that the critical parameter is the team-interaction factor, which must be kept as small as possible in order to maximize productivity (we did not need a mathematical model to tell us this, but the equation above verifies our intuition very well).

The team-production-rate equation shows very clearly what a project manager can manipulate to optimize his team's effort. If he is interested in high efficiency, he must attempt to keep $t(W)$ low by structuring the jobs into tasks which minimize the time each individual spends in interfacing his product with those of the rest of the team.

If he is interested in having a job then proceed at its highest rate, the manager must, in addition, choose the proper number of workers (with the requisite skills).

To give a simple example, suppose a manager were to divide a given programming development job, which includes design, coding, testing, documenting, quality assurance, and supervision functions into tasks such that each worker must interface with every other worker. In this case we can take

$$t(W) = (W - 1)\tau$$

where τ is the average time spent by each person interfacing with each of the others. The team production rate equation

$$P_T = P_lW[1 - (W - 1)\tau]$$

clearly has a maximum value (Fig. 1) as a function of W, at

$$W_{opt} = (1 + \tau)/2\tau \approx 1/2\tau$$

and the team efficiency at this figure is only

$$\text{efficiency} = (1 + \tau)/2 \approx 50\%$$

Adding more workers to a project already of size W_{opt} slows things down!

This behavior parallels the "maximum power transfer" law in electricity, and I refer to it as the "maximum team production rate" law: *A team producing at the fastest rate humanly possible spends half its time coordinating and interfacing.* The law holds true not just in the simple case we have assumed here, but also in any instance where $t(W)$ is roughly proportional to W. Only when this proportionality constant, τ, can be made very small does W_{opt} turn out to be so large as never to be attainable in practice.

B. Maximizing Productivity

Having identified that it is the time spent in communicating, interfacing, and integrating that lowers productivity (in this simple study), we can ask, what can be done to counteract this lessened production rate? The answer is simple in principle: *organize personnel tasks into team efforts which minimize the time individuals need to spend interfacing (and redoing their work because of improper interfacing).* Said differently: break the job into pieces which separate cleanly into parts which humans can handle easily, and whose solution then fits together well into an integrated whole.

Structured Programming (Ref. 3) using Chief Programmer Teams (Ref. 4) is one such concept which attempts to do just this for production programming. The team is divided into areas of expertise and the program is modularized so that both program and personnel interfaces are imposed top-down, in development sequence, and documented directly in the program code.

A wider concept than the Structured Programming/Chief Programmer Teams approach above, which extracts the essential features, but extends to a total development, may be described as the top-down, hierarchic, modular, structured approach to design, coding, testing, and documentation (Ref. 5). In this discipline, the design is created principally from the top-down in a modular fashion, in which each module is expanded in detail at each succeeding hierarchic level. Each design module can then be coded and tested, in that order. By making the interfaces between personnel coincide with interfaces between activities, and by making these interfaces be the *required documentation* deliverables, concurrently produced along with the program, then such interfaces tend to be totally productive, the documentation is forced to be produced and sufficient, and management has visibility into the software development process by merely monitoring the interface activity.

C. Conclusions About Productivity

The conclusion at this point, then, is that the simple one-parameter model of productivity explains the need for an organized approach in defining the development team makeup and in setting the software production discipline. It further gives one who has not yet fully appreciated the benefits of modern structured programming reason to believe that such methods can be made to work for him or for his organization.

IV. Program Readability Model

The next model is one relating to the level of documentation required for a program to be readable. Surely "readability" is a highly subjective quantity, and probably extremely variable across an ensemble of readers; but let us address the response of an "average" reader, as if there were one.

Let the program consist of L lines of compilable statements, divided into "blocks" of approximately B lines each. The uniform block size is somewhat artificial, but makes the problem easier to grasp. Let us further suppose that each block is accompanied by documentation which details its *function* i.e., a description of what the block computes, or its purpose), and also then provides additional information, which I shall call the block *rationale*. The latter contains descriptions of such things as the assumed entry and exit conditions, the significance of certain operations within the block, and relationships among data items. Such documentation may take the form of comments inserted directly into the code, or as flowcharts and narrative in some external document accompanying the code, or any other form easily understood by the intended readers.

The documentation level parameters of interest in the model are

f = the fraction of blocks having a functional description

t = the fraction of a block's total function described, when given

r = the fraction of blocks having rationale supplied

q = the fraction of a block's quantity of rationale needed, when supplied.

Let it be assumed that functional and rationale descriptions within each block can be separated so as to be independent, nonoverlapping data about what is going on in the code; that is, the functional description is to be devoid of rationale, and vice versa. This is purely a mathematical necessity for what follows, and need not be in effect in actual practice; in actuality, the two may be intermixed, and correlated in any meaningful way.

There are several durations of interest when reading a block:

T_L = time to read and comprehend the action of a line of code.

T_F = time to read and comprehend a complete function description.

T_R = time to read and comprehend a complete rationale description.

$T_{CF}(x)$ = time to create a fraction x of the missing function description needed, from fraction $1 - x$ given.

$T_{CR}(x)$ = time to create a fraction x of missing rationale description needed, from fraction $1 - x$ given.

The latter two time factors arise when there is something missing that the reader needs, in order to understand the

program. If only a fraction t of an entire functional description is given, the remaining fraction $1 - t$ must be recreated by the reader if there is to be full understanding, and the same is true concerning rationale. Both $T_{CF}(0)$ and $T_{CR}(0)$ are zero, by definition.

In order to recreate missing material, the reader may make inferences and analyses based on material supplied as part of the given block, as well as material provided outside that block. However, I will assume that a functional description should pertain entirely to the code within that block, and I shall therefore further assume that the understanding of the functional behavior of a block depends wholly on information supplied for that block. Recreating rationale may, however, require inferences based on information outside the block.

Now let us assume that if only a fraction t of the complete functional statement is given, then the time to read that functional statement is tT_F, and let similar statements describe the other reading times, as well. In such circumstances, the total average time required to read and understand a block of code will be

$$T_B = BT_L + f[tT_F + T_{CF}(1 - t)] + (1 - f)T_{CF}(1) + r[qT_R + T_{CR}(1 - q)] + (1 - r)T_{CR}(1)$$

This expression is comprised of terms, in sequence, which (1) relate the time to read the code in a block line-by-line; (2) read a functional description, when it is given; (3) and (4) recreate any missing functional information; (5) read the rationale statement, when given; and (6) and (7) recreate any missing rationale.

The same type of formula and parameters can also probably be developed to describe how long it takes one to develop a block of code in the first place, but I have not explored such an equation, as yet.

A. Documentation of Block Function

Intuitively, the time to recreate a fraction x of the functional description (by reading the code, head scratching, etc.), must increase with x—the more there is missing, the longer it takes to figure out what is going on. The curves shown in Fig. 2 represent conceptual readability indices associated with understandability of block function.

It seems intuitive that, for most computer languages, there surely must exist some minimal block size B_F such that $T_F < T_{CP}(1)$, that is, such that the time to read and understand a given complete, adequate functional description is less than the time it takes to divine that function when no such statement is provided at all. (I shall reexamine this hypothesis more a little later.) Based on such a presumption, we may conclude that for all block sizes $B > B_F$, the optimum value of f cannot be zero. That is, some fraction $f > 0$ of the blocks should always have some form of functional description, each with level of detail t. A similar reasoning shows that if blocks are larger than some value B_R, then some fraction $r > 0$ of the blocks should always have rationale supplied.

By differentiation, we can next study the sensitivity of T_B to the documentation-level parameters f, t, r, and q. First, with respect to f (the density of functional descriptions), to answer what density of the blocks should contain functional statements:

$$\frac{\partial T_B}{\partial f} = tT_F - [T_{CF}(1) - T_{CF}(1 - t)]$$

Since there is no dependency upon f in this derivative, then T_B takes its extreme values at either $f = 0$ or $f = 1$. For $B > B_F$, the value $f = 0$ has been ruled out; in addition, since there exists a value of t (namely $t = 1$) such that the derivative is negative, then T_B assuredly is maximized at $f = 1$.

That is, for every program with block size $B > B_F$, *every block should have a functional description.* Moreover, if $T'_{CF}(1 - t) = T_F$ has a solution in (0,1), there may be an optimum level of detail, $t_{opt} < 1$. Otherwise, the function should be described completely ($t_{opt} = 1$). The sensitivity of T_B to t is gauged by

$$\left.\frac{\partial T_B}{\partial t}\right|_{f=1} = T_F - T'_{CF}(1 - t)$$

The shape of $T_{CF}(x)$ is, of course, unknown; but one may speculate on its form, and conceptually, measurements could even be made to determine the characteristic. It seems reasonable that $T_{CP}(x)$ should have an increasing positive derivative; that is, that the amount of time required to figure something out should require a disproportionately longer amount of time, the more that there is to be figured out. If such were the case, and, in addition, if $T_F > T'_{CF}(0)$, then there will exist an optimum, $t_{opt} < 1$.

An optimum level of documentation detail t_{opt} less than unity (total detail) appears at that point for which an individual's reasoning facility overtakes his reading comprehension speed. It is thus advisable to leave out obvious

details and easily understood, but difficult-to-explain concepts from functional statements. Such are symptoms of overdocumentation.

B. Documentation of Block Rationale

Considerations for rationale documentation in a program run a parallel course to the functional documentation described above, but the rationale-recreation process within the reader is a different mechanism, and therefore there are some differences in the level required.

For one thing, it seems intuitive that the time required to understand the reasons for having a certain function, and the significance and relationships of block operations, variables, and data depend, to a great extent, on how well one understands the entire program surrounding the block (we have assumed function and rationale descriptions are independent within a block, but this may not necessarily hold globally).

Consequently, the time required to recreate any missing rationale needed for understanding probably depends both on q (the level of description detail in the rationale provided) as well as r (the density of blocks outside the block under scrutiny having rationale provided). In more complicated models, it probably also depends on f and t (and a number of other parameters), as well.

Investigation into proper levels for r and q is thus more intricate than the previous analysis, and, regretfully, too lengthy for inclusion here. In fact, I have not yet carried these to a point where meaningful conclusions can be drawn. However, I conjecture that the answers must be $r = 1$, $q = q_{opt} < 1$, just as was true for function: rationale should accompany *every* block larger than some B_R, and there will exist a level of detail at which reasoning rate overtakes the rate at which the volume of material needed can be read.

C. Self-documenting Programs

The reasoning above indicates that if functional blocks are too large, then every block should possess both functional and rationale descriptions. The equations also seem to indicate, in addition, that below some critical block size, no code documentation may be needed at all, other than the code itself. However, such can be true only if the program blocks are properly segmentized into understandable functional units and the rationale for and about such functions is clear, from the code statements themselves.

Is such a segmentation of a program possible? Is self-documentation of program code (in a suitable higher-level language) attainable?

The answer is probably, "No, not entirely." But again, top-down, hierarchic, modular, structured programming in a language which permits long label names goes a long way towards achieving this end. Program functional blocks can be limited to a size conducive to understanding, with a hierarchy of links to other functional (sub) blocks.

Particularly, these functional blocks can be labeled, using long enough labels, to state both the function and rationale of each given block. Contrast, for example, the following linkages to a subfunction invoked by the keyword DO; the remaining text following each DO is the subfunction label. The subfunction code is the same in each case:

```
DO  3048;
DO  S;
DO  SORT;
DO  BUBBLE SORT;
DO  BUBBLE SORT ARRAY A OF SIZE N;
DO  BUBBLE SORT ARRAY A OF SIZE N,
      BECAUSE IT IS SIMPLEST AND FAST
      ENOUGH HERE;
```

Similarly, contrast the following predicate tests, all of which refer to the same condition:

```
IF(X>Y) . . .
IF(VAL>MAX) . . .
IF(INPUT VALUE>MAXIMUM ALLOWED)
IF(INPUT IS TERMINATED AS SIGNALLED
      BY AN INPUT VALUE GREATER THAN
      THE MAXIMUM ALLOWED) . . .
.
.
.
```

In the last example, the predicate statement may be realized as a linkage to a subfunction (perhaps a macro) which computes any of the first three predicates.

Studies (Refs. 6 and 7) have indicated that the optimum block-size with respect to intellectual manageability and comprehension, contains between 5 and 10 elements. Hence, if self-documentation is attainable, the program blocks should probably be no larger than this size.

D. Conclusions About Readability

The conclusions drawn from the readability model are that, if functional blocks are too large, then they must *always* have functional and (probably) rationale descriptions. Functional blocks falling below a certain critical size have the potentiality of being self-documenting. Finally, top-down, hierarchic, modular, structured programming using long descriptive names for subfunctions and operations provides a means whereby this potential is largely achievable.

V. Program Development Model

The final model I give here is one dealing with how programs should be developed (top-down, bottom-up, inside-out, etc.) so as to minimize the programming costs. The first step toward developing the program model is to depict the program as a sequence of control graphs; each graph represents the set of program blocks at a given time as nodes with directed links to subordinate blocks (nodes), as illustrated in Fig. 3 for structured programs.

The sequence begins with G_0, the null graph, and winds up, K stages later, at $G = G_K$, the entire program graph. For convenience, I will assume that each of the stages of work in between increases the size of the graph by $1/K$-*th* of the total final program graph size; that is,

$$|G_k| - |G_{k-1}| \approx \frac{|G|}{K}$$

Now let me define the scope of control $C_k(n)$ of a node n on G_k as

$$C_k(n) = \{m | \text{ a path exists from } n \text{ to } m \text{ on } G_k\}$$

Similarly, I shall define the scope of error $S_k(n)$ of a node n on G_k as

$$S_k(n) = \{m | \text{ an error in } n \text{ requires changing } m \text{ on } G_k\}$$

In words, the scope of control of a node n is the set of all nodes whose corresponding code is connected with the evaluation of the function represented by N. The scope of error of a node n is the set of all nodes whose corresponding code must be altered, should an error be found at n.

Intuitively, it seems reasonable that $S_k(n)$, more often than not, contains $C_k(n)$, because if an error is found at n, then all subfunctions of n will probably have to be reexamined, as well as some other nodes connected to n in ways other than control. In a highly modular program, we can, in fact, probably estimate $S_k(n) \approx C_k(n)$.

A. Development Cycle

Let us now suppose that the program at stage $k-1$ has resulted in G_{k-1}, and that the extension to G_k is about to commence. Then iteratively, let a version, call it G'_k, of G_k be developed and evaluated until no errors are found, whereupon we rename G'_k as G_k, as shown in Fig. 4. Now define

$$\Delta G_k = (G_k \cup G_{k-1}) - (G_k \cap G_{k-1})$$

and separate ΔG_k into altered old code $S_k(e_1, \cdots, e_{b_k})$ and added new code, ∇G_k; the nodes $e_1 \cdots, e_{b_k}$ were found to be in error during the work stage.

Let now the cost of this stage be represented by the formula

$$\text{Cost}(k) = \$_0 |\nabla G_k| + \$_1 |S_k(e_1, \cdots, e_b)|$$

This cost presumes that both new code and alterations (in total lines of code) can be produced at uniform cost rates per program node; different cost rates apply when writing new code than repairing erroneous code.

The total cost to produce a program of N nodes under these assumptions is then

$$C_{tot} = \$_0 N + \$_1 \sum_{k=1}^{K} |S_k(e_1, \cdots e_{b_k})|$$

The number of nodes N a program contains should not be so much a function of the production method as the function that is to be performed; hence, the minimization of programming cost is effected primarily by reduction of the second term in the cost equation.

B. Error Penetration

For large programs, it seems intuitive, except for nodes near the bottom of the graph, that the magnitude of the scope of control of a node n is approximately related to the graph size in an exponential way, where the exponent is roughly the fractional number of levels between that node and the "bottom" of the subgraph dominated by n, in relation to the total number of levels in the graph. This is illustrated in Fig. 5. In a modular program, since

$S(n) \approx C(n)$, the same can probably be said about the scope of error, as well. Hence, let us define $\lambda(n)$, the *error-penetration level* of node n in the graph G_k, as

$$\lambda_k(n) = \frac{\ln|S_k(n)|}{\ln|G_k|}$$

so that we can write

$$|S_k(n)| = |G_k|^{\lambda_k(n)}$$

where $0 \leq \lambda_k(n) \leq 1$

Now since the development cycle found no errors in G_{k-1} prior to embarkation towards G_k, errors found subsequently in G_{k-1} arise mainly from assumptions made at stage $k-1$ not supportable at stage k. It is thus the addition of new nodes that has allowed the discovery of errors.

Let it therefore be assumed that the number of discovered errors is proportional to the number of added nodes (N/K) at stage n, and that the errors have nonoverlapping scopes:

$$b_k = bN/K$$

$$|S_k(e_1, \cdots, e_{b_k})| = \sum_{j=1}^{N/K} |G_k|^{\lambda_k(e_j)}$$

$$= \sum_{j=1}^{N/K} [kN/K]^{\lambda_k(e_j)}$$

Now, let it be supposed that the development process can maintain a uniform (average) error penetration throughout the implementation activity; that is, suppose $\lambda_k(e_i) \approx \lambda$. Then the total development cost would be

$$C_{tot} = \$_0 N + \$_1 \frac{bN^{1+\lambda}}{1+\lambda}[1 + 0(1/K)]$$

and is independent of K, the number of development stages, insofar as first-order effects are concerned. The cost, of course, is least when $\lambda = 0$ (zero error penetration), but I don't know of any development process that can achieve this. If N is very large, it is easy to see that the principal development costs come from the "debugging term," which could increase as rapidly as N^2, were the improper development disciplines to be in effect.

C. Minimizing Development Costs

The way to minimize the cost is to find a development procedure which maintains a uniformly low error penetration coefficient. This coefficient λ is roughly the fractional number of levels an erroneous node e lies between the top and the bottom of the subgraph controlled by e, when the scope of control can be made the same as the scope of error.

It then seems intuitive, from the above model, that one should strive to augment G_{k-1} to reach G_k by some method which adds those new nodes to G_{k-1} which have the least scopes of error inside G_{k-1} and maximum outside; one should strive also to keep the error scope within the scope of control as nearly as possible; and the starting work stage should choose G_1 as that set of nodes having the greatest scope of error in G, since these will have zero scope in G_0 (the null graph). This minimization procedure is only intuitive, I haven't actually gone through a proof that it does, in fact, minimize the cost. Such a proof is probably academic anyway, since I can't actually compute or measure scopes on *a priori* bases. However, it seems unlikely that the optimum strategy will be too different than the intuitive guidelines above.

The cost-minimizing strategy above is sometimes called "hardest-out" programming; it depends on being able to evaluate or estimate error penetration levels on *a priori* basis. Such evaluations or estimations are often possible if there is a preliminary design or baseline to work from.

If node penetration levels are not known *a priori*, then the top-down development procedure is probably the best that one can follow for several reasons. First, overall program correctness can conceivably be checked at each stage. Since the top node is the one with the greatest scope of control, it is probably then also the one with the greatest error scope. Adding nodes to the bottom of the graph at each stage keeps the scope of control of the added nodes to a minimum (zero), and thus, errors are not apt to be errors in control, but in connections between nodes other than control. If top-down structured programming is performed using a modular (in the connectivity sense) approach, then the error scopes are further reduced.

Hence, top-down hierarchic, modular, structured programming seems to be the intuitive ideal approach toward increasing programmer productivity (considering cost as being proportional to programming time) when no *a priori* estimates of error penetration levels of a program are available.

VI. Summary

I did not intend, at the outset, to make this article one extolling the virtues of Structured Programming, but rather, to illustrate that models of programming are possible and that these models can tell us something about the way programming projects should be approached for increased effectiveness. Yet it is interesting to see, at least it was for me, that in each case, Structured Programming, as discipline, was capable of delivering the optimum according to the model.

Oversimplified models can, of course, lie if they don't include enough of the real world in them. Intuitive models tend only to verify intuitive truths, and perhaps this is why Structured Programming seemed so optimum in the analyses. Did my personal bias toward Structured Programming drive me to the models and to the assumptions used in getting the solution, or vice-versa. I truthfully don't know. I do know I was trying to be objective.

I realize it is very easy to argue with oversimplifications, hypotheses, and assumptions. I realize that intuition is not proof, and that perhaps several interpretations of the results, in conflict with my own, are possible.

However, I now feel confident that basic studies, empirical measurements, statistical analyses, and applied research can someday provide less intuitive, more accurate, more sophisticated, rigorous models and simulations of the programming process. From such analyses, precise guidelines and disciplines will be discovered for improving future software developments beyond our current practical limitations, perhaps someday to their ultimate imminent theoretical maxima.

References

1. Craig, G. R., et al., "Software Reliability Study," AD-787 784, TRW Systems Group, Redondo Beach, Calif., Oct. 1974.
2. Richards, F. R., "Computer Software: Testing, Reliability Models, and Quality Assurance," AD/A-001 260, Naval Post Graduate School, Monterey, Calif., July 1974.
3. Dijkstra, E. W., Dahl, O. J., and Hoare, C.A.R., *Structured Programming*, Academic Press, New York, 1972.
4. Baker, F. T., "Chief Programmer Teams: Principles and Procedures," Report No. FSC 71-6012, IBM Corp., Gaithersburg, Md., Feb., 1972.
5. Tausworthe, R. C., *Standardized Development of Computer Software*, Jet Propulsion Laboratory, Pasadena, Calif., (to be published).
6. Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review*, Vol. 63, pp. 81–97, 1956.
7. Weinberg, G. M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Co., New York, pp. 28–40, 1971.

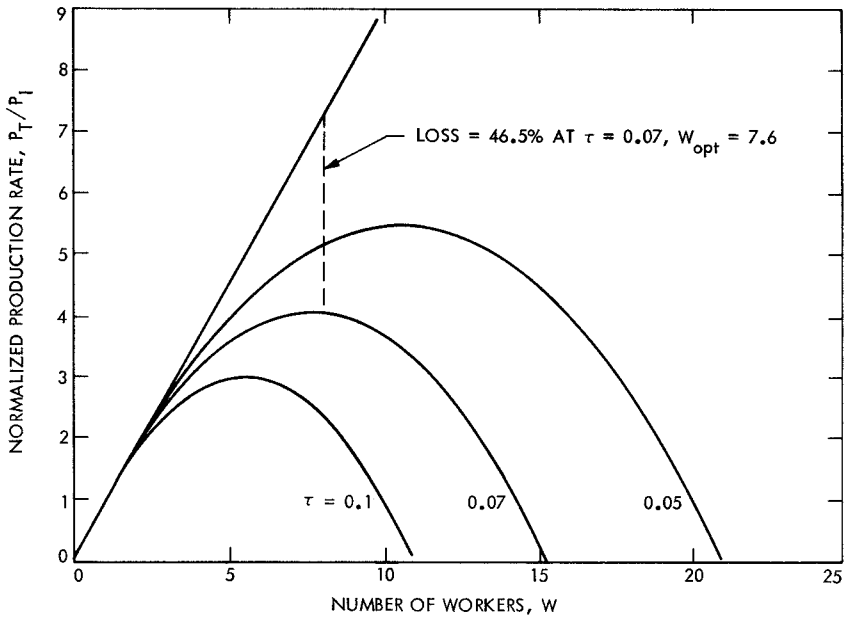

Fig. 1. Normalized team production rate as a function of team size, with communications

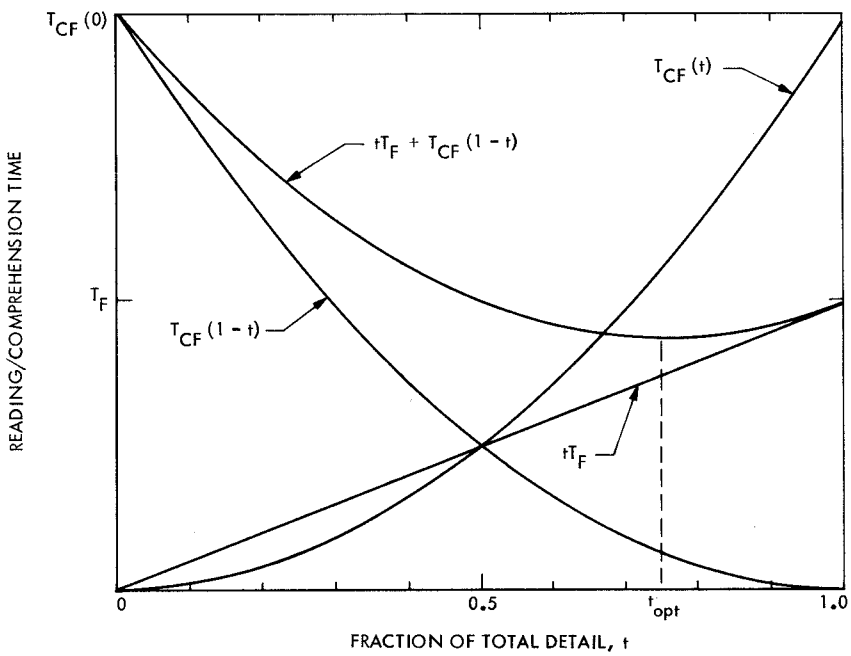

Fig. 2. Conceptual forms for readability indices for comprehension of block function

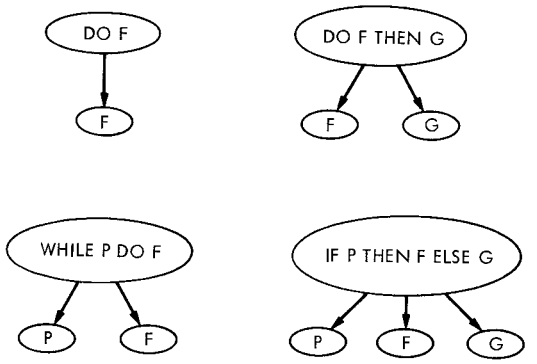

Fig. 3. Representation of a structured program as a graph in which nodes represent control connections. Each of *P*, *F*, and G, may have further expansion

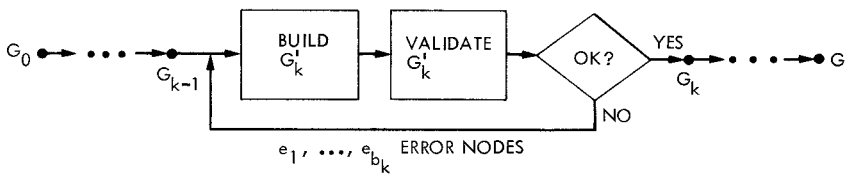

Fig. 4. Program development cycle

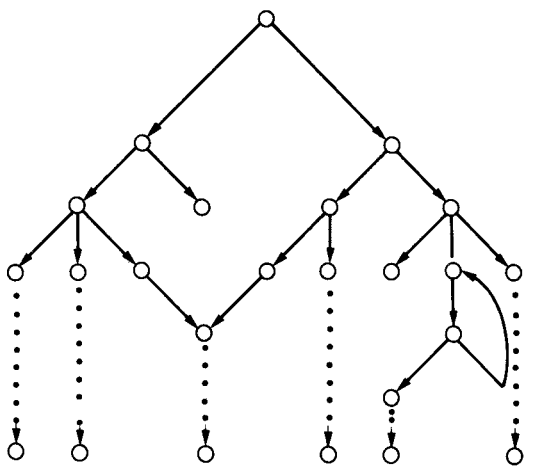

Fig. 5. Program graph which shows an almost exponential relation between node levels and the scope of control